	Type	L #	Hits	Search Text	DBs	Time Stamp	Comments
1	BRS	L1	0	(data adj flow WITH context adj flow) SAME processor SAME (concurrently or synchronously)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:20	
2	BRS	L2	0	(data adj flow WITH context adj flow) SAME processor	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:20	
3	BRS	L3	0	(data adj flow WITH context adj flow) SAME (concurrently or synchronously)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:20	
4	BRS	L4	7	(data adj flow WITH context adj flow) and (concurrently or synchronously)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:29	
5	BRS	L5	129126	(data- flow WITH context-flow) and (concurrently or synchronously)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:30	
6	BRS	L6	48616	(data- flow WITH context-flow) same (concurrently or synchronously)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:30	
7	BRS	L7	613	(data- flow WITH context-flow) same (concurrently or synchronously) SAME core	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:30	

	Type	L #	Hits	Search Text	DBs	Time Stamp	Comments
8	BRS	L8	213	(data- flow WITH context-flow) same (concurrently or synchronously) SAME core SAME control	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:31	
9	BRS	L9	52	(data- flow WITH context-flow) same (concurrently or synchronously) SAME core SAME control SAME logic	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:58	
10	BRS	L10	2	(data- flow WITH context-flow) same (concurrently or synchronously) SAME core SAME control SAME logic SAME ready SAME request	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:58	
11	BRS	L11	45	9 and signal	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:59	
12	BRS	L12	39	9 and control adj signal	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 13:59	
13	BRS	L13	6	12 and ready adj signal	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 14:00	
14	BRS	L14	4	13 not 10	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 14:50	

	Type	L #	Hits	Search Text	DBs	Time Stamp	Comments
15	BRS	L15	0	14 and data near3 flow and context near3 flow	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 14:51	
16	BRS	L16	5	14 and data- flow and context-flow	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 14:58	
17	BRS	L17	1	16 and context	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 14:56	
18	BRS	L18	0	14 and " data- flow" and "context-flow"	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 14:58	
19	BRS	L19	1	" data- flow" and "context-flow"	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/24 14:58	

	Type	L #	Hits	Search Text	DBs	Time Stamp	Comments
1	BRS	L1	0	data SAME context SAME processing SAME core SAME storage SAME regiter SAME multiplexer	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/22 13:10	
2	BRS	L2	1	data SAME context SAME processing SAME core SAME storage SAME register SAME multiplexer	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/22 13:11	
3	BRS	L3	36	data SAME context SAME processing SAME core SAME storage	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/22 13:13	
4	BRS	L4	7	3 and register and multiplexer	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/22 13:13	
5	BRS	L5	166	data SAME context SAME processing SAME core and storage	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/22 13:13	
6	BRS	L6	52	5 and register and multiplexer	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/22 13:13	
7	BRS	L7	45	6 not 4	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/22 13:14	

PORTAL
US Patent & Trademark Office

Subscribe (Full Service) Register (Limited Service, Free) Login

Search: ● The ACM Digital Library ○ The Guide

core and (conurrently or synchronously) and context and data

THE ACM DIGITAL LIBRARY

Feedback Report a problem Satisfaction survey

Terms used
core and **conurrently** or **synchronously** and **context** and **data**

Found **44,723** of **134,837**

Sort results by: relevance

Display results: expanded form

Save results to a Binder

Search Tips

□ Open results in a new window

Try an Advanced Search
Try this search in The ACM Guide

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next

Best 200 shown

Relevance scale

1 Session P2: large data sets: Out-of-core rendering of massive geometric environments
Gokul Varadhan, Dinesh Manocha
October 2002 **Proceedings of the conférence on Visualization '02**

Full text available: pdf(2.38 MB) Additional Information: full citation, abstract, references, citings, index terms

We present an external memory algorithm for fast display of very large and complex geometric environments. We represent the model using a scene graph and employ different culling techniques for rendering acceleration. Our algorithm uses a parallel approach to render the scene as well as fetch objects from the disk in a synchronous manner. We present a novel prioritized prefetching technique that takes into account LOD-switching and visibility-based events between successive frames. We have appli ...

Keywords: LODs, external memory, large datasets, prefetching, visibility, walkthroughs

2 Fast detection of communication patterns in distributed executions
Thomas Kunz, Michiel F. H. Seuren
November 1997 **Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research**

Full text available: pdf(4.21 MB) Additional Information: full citation, abstract, references, index terms

Understanding distributed applications is a tedious and difficult task. Visualizations based on process-time diagrams are often used to obtain a better understanding of the execution of the application. The visualization tool we use is Poet, an event tracer developed at the University of Waterloo. However, these diagrams are often very complex and do not provide the user with the desired overview of the application. In our experience, such tools display repeated occurrences of non-trivial commun ...

3 Context-based prefetch – an optimization for implementing objects on relations
Philip A. Bernstein, Shankar Pal, David Shutt
December 2000 **The VLDB Journal — The International Journal on Very Large Data Bases**, Volume 9 Issue 3

Full text available: pdf(142.24 KB) Additional Information: full citation, abstract, index terms

When implementing persistent objects on a relational database, a major performance issue is prefetching data to minimize the number of round-trips to the database. This is especially hard with navigational applications, since future accesses are unpredictable. We propose the use of the context in which an object is loaded as a predictor of future accesses, where a context can be a stored collection of relationships, a query result, or a complex object. When an object O's state is loaded, similar ...

Keywords: Caching, Object-oriented database, Object-relational mapping, Prefetch

4 Some Major Issues in the Design of the Core Graphics System
James C. Michener, James D. Foley
December 1978 **ACM Computing Surveys (CSUR)**, Volume 10 Issue 4
Full text available: pdf(1.25 MB) Additional Information: full citation, references, citings, index terms

5 Distributed file systems: concepts and examples
Eliezer Levy, Abraham Silberschatz
December 1990 **ACM Computing Surveys (CSUR)**, Volume 22 Issue 4
Full text available: pdf(5.33 MB) Additional Information: full citation, abstract, references, citings, index terms, review

The purpose of a distributed file system (DFS) is to allow users of physically distributed computers to share data and storage resources by using a common file system. A typical configuration for a DFS is a collection of workstations and mainframes connected by a local area network (LAN). A DFS is implemented as part of the operating system of each of the connected computers. This paper establishes a viewpoint that emphasizes the dispersed structure and decentralization of both data and con ...

6 Extensibility safety and performance in the SPIN operating system
B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, S. Eggers
December 1995 **ACM SIGOPS Operating Systems Review , Proceedings of the fifteenth ACM symposium on Operating systems principles**, Volume 29 Issue 5
Full text available: pdf(2.32 MB) Additional Information: full citation, references, citings, index terms

7 Data integration and sharing II: Scientific data repositories: designing for a moving target
Etzard Stolte, Christoph von Praun, Gustavo Alonso, Thomas Gross
June 2003 **Proceedings of the 2003 ACM SIGMOD international conference on on Management of data**
Full text available: pdf(739.27 KB) Additional Information: full citation, abstract, references, index terms

Managing scientific data warehouses requires constant adaptations to cope with changes in processing algorithms, computing environments, database schemas, and usage patterns. We have faced this challenge in the RHESSI Experimental Data Center (HEDC), a datacenter for the RHESSI NASA spacecraft. In this paper we describe our experience in developing HEDC and discuss in detail the design choices made. To successfully accommodate typical adaptations encountered in scientific data management systems ...

8 Middleware for context sensitive mobile applications
K. A. Hawick, H. A. James
January 2003 **Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003 - Volume 21**
Full text available: pdf(916.67 KB) Additional Information: full citation, abstract, references, index terms

Contextual information such as spatial location can significantly enhance the utility of mobile applications. We introduce the concept of active preferences that represent a combination of user preference information and choices combined with spatial or temporal information. Active preferences set the policy on how a mobile application should customise its behaviour not just for a particular user but as that user moves to different locations and interacts with other mobile users or with fixed lo ...

Keywords: location context, middleware, mobile devices, personal context

9 Practicing JUDO: Java under dynamic optimizations
Michał Cierniak, Guei-Yuan Lueh, James M. Stichnoth
May 2000 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation**, Volume 35 Issue 5

Full text available: pdf(190.06 KB) Additional Information: full citation, abstract, references, citings, index terms

A high-performance implementation of a Java Virtual Machine (JVM) consists of efficient implementation of Just-In-Time (JIT) compilation, exception handling, synchronization mechanism, and garbage collection (GC). These components are tightly coupled to achieve high performance. In this paper, we present some static anddynamic techniques implemented in the JIT compilation and exception handling of the Microprocessor Research Lab Virtual Machine (MRL VM), ...

10 Real-time convergence of Ada and Java™
Ben Brosgol, Brian Dobbing
September 2001 **ACM SIGAda Ada Letters , Proceedings of the 2001 annual ACM SIGAda international conference on Ada**, Volume XXI Issue 4

Full text available: pdf(191.98 KB) Additional Information: full citation, abstract, references, citings, index terms

Two independent recent efforts have defined extensions to the Java platform that intend to satisfy real-time requirements. This paper summarizes the major features of these efforts, compares them to each other and to Ada 95's Real-Time Annex, and argues that their convergence with Ada95 may serve to complement rather than compete with Ada in the real-time domain.

Keywords: Ada, Java, Real-Time, asynchrony, garbage collection, scheduling, threads

11 Video Storage: System support for providing integrated services from networked multimedia storage servers
Ravi Wijayaratne, A. L. Narasimha Reddy
October 2001 **Proceedings of the ninth ACM international conference on Multimedia**

Full text available: pdf(227.49 KB) Additional Information: full citation, abstract, references, index terms

In this paper, we describe our experience in building an integrated multimedia storage system, Prism. Our current Linux-based implementation of Prism provides three levels of service: deadline guarantees for *periodic* applications, best-effort better response times for *interactive* applications and starvation-free throughput guarantees for *aperiodic* applications. Prism separates resource allocation from resource scheduling. Resource allocation is controlled across the service ...

Keywords: admission control, disk, file systems, multimedia, scheduling

12 A survey of structured and object-oriented software specification methods and techniques
Roel Wieringa
December 1998 **ACM Computing Surveys (CSUR)**, Volume 30 Issue 4

Full text available: pdf(605.26 KB) Additional Information: full citation, abstract, references, citings, index terms, review

This article surveys techniques used in structured and object-oriented software specification methods. The techniques are classified as techniques for the specification of external interaction and internal decomposition. The external specification techniques are further subdivided into techniques for the specification of functions, behavior, and communication.

After surveying the techniques, we summarize the way they are used in structured and object-oriented methods and indicate ways in w ...

Keywords: languages

13 High-performance sorting on networks of workstations
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, David A. Patterson
June 1997 **ACM SIGMOD Record , Proceedings of the 1997 ACM SIGMOD international conference on Management of data**, Volume 26 Issue 2
Full text available: pdf(1.53 MB) Additional Information: full citation, abstract, references, citings, index terms

We report the performance of NOW-Sort, a collection of sorting implementations on a Network of Workstations (NOW). We find that parallel sorting on a NOW is competitive to sorting on the large-scale SMPs that have traditionally held the performance records. On a 64-node cluster, we sort 6.0 GB in just under one minute, while a 32-node cluster finishes the Datamation benchmark in 2.41 seconds. Our implementations can be applied to a variety of disk, memory, and processor configura ...

14 A design space for multimodal systems: concurrent processing and data fusion
Laurence Nigay, Joëlle Coutaz
May 1993 **Proceedings of the SIGCHI conference on Human factors in computing systems**
Full text available: pdf(806.88 KB) Additional Information: full citation, abstract, references, citings, index terms

Multimodal interaction enables the user to employ different modalities such as voice, gesture and typing for communicating with a computer. This paper presents an analysis of the integration of multiple communication modalities within an interactive system. To do so, a software engineering perspective is adopted. First, the notion of "multimodal system" is clarified. We aim at proving that two main features of a multimodal system are the concurrency of processing and the fusion ...

Keywords: concurrency, data fusion, design space, modality, multimodal interaction, software architecture, taxonomy

15 The convergence of AOP and active databases: towards reactive middleware
Mariano Cilia, Michael Haupt, Mira Mezini, Alejandro Buchmann
September 2003 **Proceedings of the second international conference on Generative programming and component engineering**
Full text available: pdf(330.81 KB) Additional Information: full citation, abstract, references, index terms

Reactive behavior is rapidly becoming a key feature of modern software systems in such diverse areas as ubiquitous computing, autonomic systems, and event-based supply chain management. In this paper we analyze the convergence of techniques from aspect oriented programming, active databases and asynchronous notification systems to form reactive middleware. We identify the common core of abstractions and explain both commonalities and differences to start a dialogue across community boundaries. W ...

16 Virtual memory primitives for user programs
Andrew W. Appel, Kai Li
April 1991 **Proceedings of the fourth international conference on Architectural support for programming languages and operating systems**, Volume 26 , 19 , 25 Issue 4 , 2 , Special Issue
Full text available: pdf(1.37 MB) Additional Information: full citation, references, citings, index terms

17 A design space for multimodal systems: concurrent processing and data fusion
Laurence Nigay, Joëlle Coutaz
January 1993 **Proceedings of the conference on Human factors in computing systems**
Full text available: pdf(851.14 KB) Additional Information: full citation, references, index terms

Keywords: concurrency, data fusion, design space, modality, multimodal interaction, software architecture, taxonomy

18 Special session on reconfigurable computing: Reconfigurable platforms for ubiquitous computing
Manfred Glesner, Thomas Hollstein, Leandro Soares Indrusiak, Peter Zipf, Thilo Pionteck, Mihail Petrov, Heiko Zimmer, Tudor Murgan
April 2004 **Proceedings of the first conference on computing frontiers on Computing frontiers**
Full text available: pdf(479.97 KB) Additional Information: full citation, abstract, references, index terms

Ubiquitous computing requires flexibilty. Melting distributed electronic devices into everyday's life implies the need to adapt to evolving standards and dynamic environments. Furthermore, to gain user acceptance, such devices should be able to adapt to different usage patterns and user profiles. Scalability is also an important issue, allowing functional enhancements to already deployed systems. In this work we address these issues applying the concept of reconfigurability on different abstract ...

Keywords: communication, dynamic power management, networks-on-chip, reconfigurable hardware, reconfigurable processors, reconfiguration, ubiquitous computing

19 The many faces of publish/subscribe
Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, Anne-Marie Kermarrec
June 2003 **ACM Computing Surveys (CSUR)**, Volume 35 Issue 2
Full text available: pdf(456.46 KB) Additional Information: full citation, abstract, references, index terms

Well adapted to the loosely coupled nature of distributed interaction in large-scale applications, the publish/subscribe communication paradigm has recently received increasing attention. With systems based on the publish/subscribe interaction scheme, subscribers register their interest in an event, or a pattern of events, and are subsequently asynchronously notified of events generated by publishers. Many variants of the paradigm have recently been proposed, each variant being specifically adap ...

Keywords: Distribution, interaction, publish/subscribe

20 Virtual Memory
Peter J. Denning
September 1970 **ACM Computing Surveys (CSUR)**, Volume 2 Issue 3
Full text available: pdf(2.63 MB) Additional Information: full citation, references, citings, index terms

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next

The VLDB Journal (2000) 9: 177–189

The VLDB Journal
© Springer-Verlag 2000

Context-based prefetch – an optimization for implementing objects on relations

Philip A. Bernstein, Shankar Pal, David Shutt

Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA; E-mail: {philbe,shankarp,dshutt}@microsoft.com

Edited by M.P. Atkinson. Received May 3, 2000 / Accepted October 26, 2000

Abstract. When implementing persistent objects on a relational database, a major performance issue is prefetching data to minimize the number of round-trips to the database. This is especially hard with navigational applications, since future accesses are unpredictable. We propose the use of the context in which an object is loaded as a predictor of future accesses, where a context can be a stored collection of relationships, a query result, or a complex object. When an object O's state is loaded, similar state for other objects in O's context is prefetched. We present a design for maintaining context and for using it to guide prefetch. We give performance measurements of its implementation in Microsoft Repository, showing up to a 70% reduction in running time. We describe several variations of the optimization: selectively applying the technique based on application and database characteristics, using application-supplied performance hints, using concurrent database queries to support asynchronous prefetch, prefetching across relationship paths, and delayed prefetch to save database round-trips.

Key words: Prefetch – Caching – Object-relational mapping – Object-oriented database

1 Introduction

One way to implement persistent objects is to map them to a relational database system (RDBMS). This approach has two main benefits: it provides persistent object views of existing relational databases; and it allows an RDBMS customer to build new object-oriented databases without introducing a new database engine, thereby avoiding changes to database administration procedures and interoperability problems with existing applications. The approach is even more attractive with object-relational DBMSs, which typically support more of the desired object functionality in the database engine itself. The main disadvantage of mapping objects to relations is performance, which for many common usage scenarios is well below that of object-oriented database systems (OODBs) that use storage servers designed explicitly for object-oriented access.

An important feature of persistent object implementations (on any kind of storage system) is the ability to load persistent objects as active main memory objects, using the object model of the application environment (e.g., C++, Java, Smalltalk, OMG CORBA, or COM). This minimizes the impedance mismatch between the language and DBMS [11], but creates performance challenges for the database implementation, especially when mapped to an RDBMS rather than a custom storage system.

One major performance problem is that application object models are inherently navigational. That is, objects have references or relationships to other objects, which applications follow one at a time. Caching of recently-accessed objects is helpful to avoid accessing the RDBMS too often. But even with caching, if each access to a non-cached object entails a round-trip to the RDBMS, performance will be unbearably slow.

To get a feeling for the performance penalty of round-trips to an RDBMS, consider the following simple experiment: Define a relational database consisting of one table, whose 100 000 rows are 100 bytes each. Each row has a 16-byte ObjectID column which has a clustered index, three 24-byte string-valued columns, and three 4-byte integer-valued columns. Suppose the application knows which ObjectID values it wants, and it retrieves the rows for 100 randomly selected keys in batches of 1, 20, or 100 rows. Using a warm server cache to factor out the cost of disk accesses, we ran the experiment on an RDBMS product and retrieved 580 rows per second, 2700 rows per second, and 3200 rows per second for batch sizes 1, 20, and 100 respectively.[1] This corresponds to a retrieval time of 170 ms, 37 ms, and 31 ms for 100 rows. In

This is an extended version of the paper "Context-Based Prefetch for Implementing Objects on Relations," which appeared in the 25th VLDB Conference Proceedings.

[1] All experiments in this paper use commodity hardware. The hardware and software configurations are left unspecified, to avoid the usual legal and competitive problems with publishing performance numbers for commercial products. All performance measurements are averages over multiple trials, with more trials for higher variance measurements.

this case, it is up to 5.5 times faster to get rows (i.e., object states) in a batch of 100 rows than a row at a-time.

To minimize the performance penalty of database round-trips, applications often issue a query to identify the objects of interest and then scan the resulting cursor, one object at a time. This tells the DBMS which objects to retrieve in batch, but often leaves open which pieces of the objects' state are desired, and hence which tables to access. Moreover, for many simple popular navigational patterns, such as following a relationship, it's a nuisance to issue a query. The application programmer would be happier to navigate from one object to the next, accessing the objects as she needs them, letting the DBMS automatically determine what data to prefetch. Programming interfaces for most OODBs, such as the ODMG model [7], satisfy this desire by offering both navigational and query access. But how can the DBMS figure out what to prefetch in response to navigational access? This is the central question addressed by this paper.

Some OODBs use page servers [17,20]. When accessing an object, the page-oriented OODB (POODB) retrieves the page containing the object, and therefore prefetches other data on the same page. Thus, by clustering data that will be accessed together on the same page, a POODB ensures effective prefetching. That is, the clustering approach amounts to a static prefetching algorithm. Clustering and prefetching are dual problems.[2]

Suppose that whenever a navigational application gets an object O for the first time, it shortly thereafter accesses most of the objects on O's page – the best case for a POODB, whose performance in this case is hard to beat. By comparison, when implementing objects on an RDBMS, there is an additional query processing step to find the same records that the POODB clustered on O's page. Even if the RDBMS clusters the records the same way, it still takes more time to find them and gather them up for transmission to the application than the POODB, which simply ships the page.

Even if the POODB's clustering strategy is optimal for the average workload, access patterns vary and the page-oriented prefetching will make mistakes. Sometimes it will make useless prefetches, where the fetched data isn't subsequently accessed. At other times it will miss prefetch opportunities, because a predictable access pattern hits objects that are mostly on different pages. These mistakes are inherent in the architecture: static clustering of records and page-oriented accesses.

A system where objects are mapped to an RDBMS (we'll call it an *OMRDB*) probably cannot match the POODB for access patterns that follow the POODB's physical data clustering. However, it may be able to earn back some of that lost performance, in the following two ways: First, since the OMRDB uses a row server, not a page server, it can prefetch an arbitrary combination of rows. That is, it can use knowledge of recent application behavior to identify combinations of rows worth prefetching, and then use its powerful query processor to find and retrieve those rows in one round-trip, whether or not the data is physically clustered. By contrast, a POODB generally retrieves pages from the server on demand. For a given data layout, both OMRDB and POODB will retrieve the same number of pages. But query-based prefetching allows an OMRDB to prefetch rows before they're referenced, reducing latency. Second, if the density of desired rows on each page is low, then the OMRDB will make better use of client cache than a POODB (since it prefetches only desired rows) and will use less network bandwidth to transfer prefetched rows. Of course, OODBs that use object servers rather than page servers can use this OMRDB tactic, so they too can benefit from the techniques described in this paper.

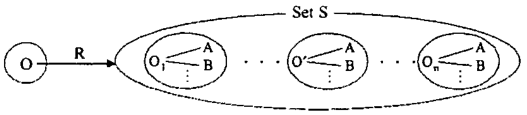

Fig. 1. Simple example of context-based prefetch

When prefetching objects, an OMRDB has two related decisions to make: which objects to prefetch and which portions of those objects' state to prefetch. To illustrate our technique, we focus first on the latter question with a simple example (see Fig. 1). Suppose an application accesses relationship R on object O, which returns a set of objects, S. Suppose the state of each object in S is spread across multiple tables. The application may not access all of that state of each object. To avoid prefetching a state that the application does not need, the OMRDB delays the decision of which state to prefetch. Instead, it simply retrieves the object IDs of the objects in S (making a round-trip to the RDBMS) and waits to see what the application does next. Suppose the application selects an object O′ in S (which is now in application cache) and accesses attribute A in O′. This requires another round-trip. But rather than just retrieving A for O′, the OMRDB retrieves (prefetches) A for all objects in S. This is useful if the application later accesses A for many of those other objects in S, a very common access pattern in workloads we have observed.

Notice that the prefetch decision is based on the application's access pattern – it is not statically determined. O′ could be a member of many collections in the database. The decision to prefetch A for all objects in S is based on the fact that O′ was fetched as part of S, and not some other collection. The OMRDB must remember this fact in order to use S as the basis for prefetching A. This is the core idea of our prefetching technique: The OMRDB uses the context in which each object is accessed as a predictor of other objects that will be accessed later.

To implement this approach, the OMRDB creates a *structure context* for each object, which describes the structure in which the object was fetched. Examples of "structures" are stored collections of relationships, query results, and complex objects. When accessing some state of an object O, the OMRDB prefetches the same pieces of the state for other objects in the structure context of O, as in the example of accessing the attribute A of O′ for all objects in the structure context S of O′. We call this approach *context-based prefetch*. As in the above example, the approach is beneficial whenever all objects in a context undergo similar manipulation.

The rest of the paper is organized as follows. Section 2 presents the basic mechanisms for context-based prefetch. Since the technique is not cost effective in all situations, Sect. 3 proposes performance hints to selectively enable the optimizations. A summary of our implementation in Microsoft[3] Repos-

[2] As discussed in the Conclusion section of [12].

[3] Microsoft is a trademark of Microsoft Corporation.

itory version 2.0 (in Microsoft SQL Server 7.0) is discussed in Sect. 4. Performance measurements in Sect. 5 show up to a 3-fold speedup due to these optimizations. Section 6 describes extensions for asynchronous prefetch, lazy loading of objects, and prefetching across paths. Section 7 summarizes related work on the general problem of implementing OMRDBs efficiently and compares our technique to other published optimization approaches. Section 8 is the conclusion.

2 Using structure context

2.1 Object model

To describe details of the approach, we need to define an object model. We choose one that is similar to those in common use, such as the ODMG model [7], COM [30], and UML [3, 29]. The approach is largely insensitive to the details of the model used here. It should work well for any model that groups objects into structures.

Each persistent object has a persistent state that consists of attributes. Each attribute value can be a *scalar*, an *object*, or a *set*.

- Each scalar-valued attribute conforms to a *scalar type*, which gives the name of the attribute and its data type, such as string, integer, or Boolean.
- Every object has a scalar-valued ObjectID attribute that uniquely identifies the object.
- Each object-valued attribute is one side of a *binary relationship*. That is, each relationship consists of two objects that refer to each other.
- Each set-valued attribute contains an object of type set, which in turn contains a set of either scalar values or object references. The concept of set is a representative example of a generic structure type. Other structure types would be handled analogously to sets, such as sequence, array, table, or record structure, but we do not consider them here.

Each object conforms to an object type. Each *object type* has a name and a set of attribute types it can contain. Each binary relationship conforms to a *relationship type*, which gives the name of the relationship, the two object types that can be related, and for each of the two object types, the attribute name by which the reference is accessed.

A *class* is a body of code that implements one or more object types. It includes a class factory that produces objects that are instances of the class. It also includes code that implements the usual read and write operations on all of the attributes and structures of the object types that the class implements.

2.2 Operations

The set of navigational object-oriented operations that we consider are GetObject, GetAttribute, GetNext, and ExecuteQuery. These are meant to be a representative sample of the kinds of navigational operations found in programming interfaces for persistent object systems.

- GetObject(ObjectID) returns a running copy of the persistent object whose unique identifier is ObjectID.

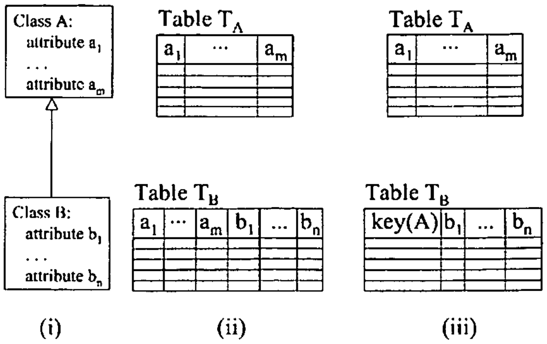

Fig. 2. Mapping classes to tables

- O.GetAttribute(AttrName) returns the value of the attribute AttrName from object O. (The notation O.M means execute method M on object O.) The result is a scalar, object, or set, depending on the attribute type. A set has an associated cursor, initially pointing to the set's first element.
- S.GetNext either returns the scalar or object identified by set S's cursor and advances the cursor, or, if the cursor points beyond the end of S, returns null.
- ExecuteQuery(Q) returns the set of objects that satisfy query Q's qualification (as in OQL [7,10]).

2.3 Database schema

An OMRDB maps objects to rows of tables. A class maps to a table whose columns represent its single-valued attributes. Our optimizations are applicable independent of the rules used to map attributes of a class to a particular table. However, for completeness, we give a few details of mappings that are commonly used.

The simplest mapping is to map a class to exactly one table that contains all of the class's attributes. But more complex mappings are also popular. For example, suppose class B inherits from a class A. If both classes are concrete (i.e., have instances), then there are separate tables for B and A. A's columns, which B inherits, may be stored in both A's and B's tables (Fig. 2(ii)), or only in A's table (Fig. 2(iii), sometimes called "vertical partitioning" [16]). In the latter case, B's state is reconstructed by joining A's and B's tables. If A is abstract, then its columns might only be stored in tables of concrete classes that inherit from it, in which case it has no corresponding table (i.e., in Fig. 2(ii), only store Table T_B, sometimes called "horizontal partitioning").

We assume each many-to-many relationship type is represented in a "junction" table. There could be a separate junction table for each relationship type, with columns SourceObjectID and TargetObjectID, or a generic junction table for all relationship types with columns SourceObjectID, RelationshipTypeID, and TargetObjectID. Each one-to-many relationship can be represented either in a junction table or as a foreign key on the "many side." For example, if the one-to-many is parent-child, then the foreign key to the parent is stored in the child.

An attribute consisting of a set of scalars can be stored in a table with columns ObjectID, AttributeID (short form of the attribute name), and Value. If the set's maximum cardinality N is known, it could instead be stored as columns of the class's table, such as $\text{AttrName}_1, \ldots, \text{AttrName}_N$. Since these two table structures are isomorphic to one-to-many relationships and single-valued attributes (respectively), the prefetch scenarios for a set of scalar attributes are isomorphic to those other two cases as well and therefore are not treated further in this paper.

2.4 *The prefetch pattern*

As discussed in Sect. 1, the main approach is to maintain a structure context (or, more simply, a *context*) for each object, which describes the structure in which the object was loaded, and to use that context to guide later prefetch decisions. In this section, we explain one usage of the approach in detail. We reapply this usage to other operations in the next section.

Consider the following operation sequence:

a. S = O.GetAttribute(R), which returns a set S of objects, which is the value of relationship attribute R.
b. O′ = S.GetNext, which returns an object O′ in S.
c. V = O′.GetAttribute(A), which returns the value V of scalar-valued attribute A of O′.

This is the scenario of Fig. 1. Attribute A corresponds to a column of a table, T, containing (some of) the state of the class, C, of O′. Unless T is very wide (e.g., has lots of long columns), it costs little more to retrieve all of T's columns that are part of C's state than to retrieve only A. This is because most of the cost is in the disk accesses, which retrieve all of the columns from disk whether or not they are fetched by the OMRDB. Thus, if there is a good chance that some of those columns will be accessed, then it is worth prefetching all of those columns of T for O′. Moreover, since O′ was retrieved via S, if we expect other objects in S to be accessed similarly to O′, then we should prefetch all of those columns of T for all objects in S, not just for O′. This avoids later round-trips to the database for each object in S.

The optimization is illustrated in Fig. 3. Table T is shown in Fig. 3(i), with relationships, such as R, implemented by a junction table J. Steps (a) and (c) above are illustrated in Fig. 3(ii) and (iii) respectively. Notice that Step (c) uses the same selection clauses for J as Step (a), and then joins with T to get the columns of T for *all* objects in O.GetAttribute(R), not just for O′.

The experiment in Sect. 1 suggests this prefetch is profitable if at least 4-5 items in the collection are later accessed (since batch retrieval is 4-5 times the cost of a single-row retrieval). However, the addition of object-level processing reduces the fractional contribution of DB round-trips to total cost. Thus, in our experiments, across a variety of workloads and database profiles, the prefetch is profitable if at least 3 items are subsequently accessed (for our combination of data server, network, etc.). The more items accessed, the more round-trips saved by the prefetch, and hence the greater the benefit.

If objects in S have state in two tables, T and T′, is it worth getting attributes from both of them? Usually not. For example, we extended the experiment in Sect. 1 by duplicating the table, to model T and T′. Getting 100 rows from both tables in one round-trip added 50% to the execution time over retrieving only from T. So the cost of needlessly getting the columns of T′ is high. Also, the benefit is modest, since retrieving those rows from both tables was only 19% slower in two round-trips than retrieving the join in one round-trip. Thus, one should only retrieve the columns of T′ if they are almost certain to be needed.

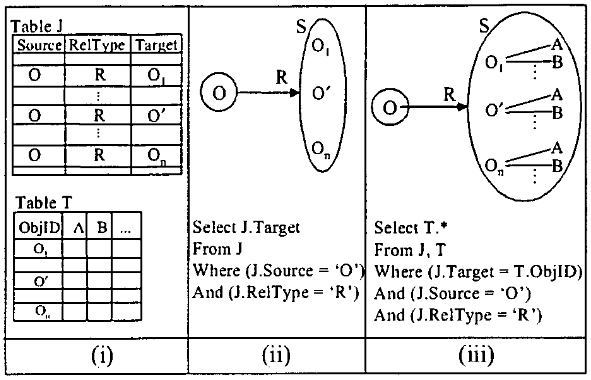

Table J

Source	RelType	Target
O	R	O_1
	⋮	
O	R	O′
	⋮	
O	R	O_n

Table T

ObjID	A	B	...
O_1			
O′			
O_n			

```
Select J.Target
From J
Where (J.Source = 'O')
And (J.RelType = 'R')
```

```
Select T.*
From J, T
Where (J.Target = T.ObjID)
And (J.Source = 'O')
And (J.RelType = 'R')
```

Fig. 3. Prefetching scenario

Since the technique heavily uses cache, the interactions of prefetching and cache management need careful attention. For example, if the cache is nearly full or if the data to prefetch is very large, then the prefetch may not work well. We will see other examples of this later.

To generate the SQL query shown in Fig. 3(iii), the OMRDB needs the context of O′. The context should include the information that was used to create the set S initially, namely, the object ID of the relationship's source, O, and the relationship name, R. These are the parameters that are needed to construct the SQL query, which retrieves the desired attributes of all objects in the context.

We expect it is worth supporting variations of this pattern where only attribute A or a predefined subset of attributes is retrieved, and not all the attributes of its table. However, as this is only a slight variation of our main idea, we do not consider it further in this paper.

2.5 *Case analysis*

We generalize Sect. 2.4 to other navigational access patterns that suggest future navigational behavior. These suggestions lead naturally to prefetch recommendations, which retrieve the data needed to service the later accesses before those accesses occur. Of course, recent navigational accesses don't guarantee that those later accesses will occur, so prefetch recommendations must be applied selectively, an issue we will discuss in Sect. 3. For now, we simply describe potentially useful prefetches and how to implement them, for each of the operation types in Sect. 2.2.

In the following descriptions, we omit the initial test to determine if the requested data is already in cache and therefore does not need to be fetched.

GetObject(ObjectID) – Prefetch some or all of the object's state. Note that to load the object into main memory, the OMRDB only needs to know the object's class (to know what class

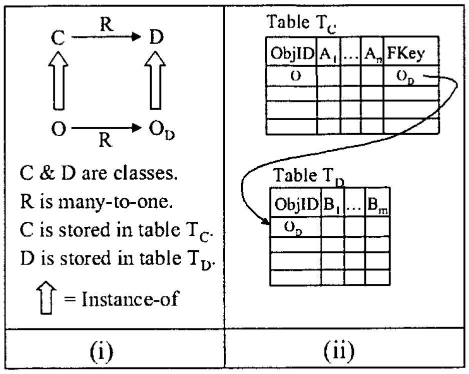

Fig. 4. Prefetching attributes with a foreign key

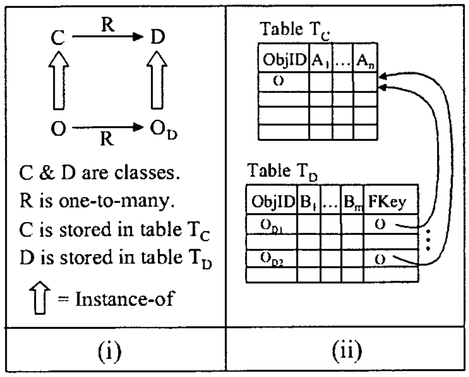

Fig. 5. Prefetching attributes of referenced objects

to instantiate) and that the object exists (to know whether to return an error). Prefetching other object states is optional at this stage. Set the object's context to null, which means it was loaded directly, not as part of a larger structure.

O.GetAttribute(AttrName) – where AttrName is the name of an attribute in O's class. There are six cases to consider, Case (i) – Case (vi) below:

Case (i) If AttrName is a single-valued object reference, and O's context is null, then get the reference to the object from the database. If the object reference is stored as a foreign key in a table containing other attributes of O, then retrieve those other attributes too.

For example, suppose AttrName is a many-to-one relationship R from O's class C to class D (see Fig. 4). Suppose C's table T_C(ObjID, $A_1, \ldots, A_n$, FKey) contains scalar attributes $A_1, \ldots, A_n$ of C and foreign key attribute FKey that contains the object ID of an object in D related via R. Since getting the reference to O_D in D involves accessing the FKey column of O's row in T_C, prefetch the other attributes $A_1, \ldots, A_n$ of that row too.

If T_C is indexed on the compound key [ObjID, FKey], rather than just ObjID, then the query processor can get O_D without accessing O's row, making it cheaper to retrieve FKey by itself. If the index on [ObjID, FKey] is non-clustered, this raises the incremental cost of getting attributes $A_1, \ldots, A_n$, which should therefore be prefetched too only if there's a high probability they will subsequently be accessed.

Case (ii) If AttrName is a scalar and O's context is null, then simply retrieve the attribute. As in Case (i), retrieve other attributes of O's state that are in the table containing AttrName's column. Here and in Case (i), prefetching those other attributes is cost-beneficial if some of them are subsequently referenced.

Case (iii) If AttrName is a single-valued object reference, and the context of O is a set S′, then prefetch the reference for every object O′ in S′, not just for O. That is, run one SQL statement that returns the ObjectID value of AttrName for all objects in S′. As in Case (i), if the object references are stored as foreign keys in a table containing other attributes of O′, then retrieve those other attributes too. Prefetching the object reference for other objects in S′ is cost-beneficial if at least several other objects in S′ are subsequently accessed. This case is essentially the same as Case (i) (Fig. 4), except that multiple rows of T_C are retrieved (one for each object in S′) instead of just one (for O).

Case (iv) If AttrName is a scalar and the context of O is a set of objects, then O.GetAttribute(AttrName) corresponds to Step (c) in Sect. 2.4. Do the prefetch described there and in Fig. 3(iii).

Case (v) If AttrName is set-valued and the context of O is null, then retrieve the set's content for O, which is a set S of either scalar values or objects. In the latter case, assign S to be the context of each object in S.

Suppose S is a set of objects (as opposed to scalars). If AttrName is a one-to-many relationship, then the references from O to objects in S can be stored as O's foreign key in the objects in S. So, when accessing AttrName, retrieve other attributes of the objects in S that are stored in the same table as the foreign key. Modifying the example of Fig. 4 so that AttrName is a one-to-many (rather than many-to-one) relationship R from O's class C to class D, we get Fig. 5, where $B_1, \ldots B_m$ can be retrieved along with objects in D referenced by O. As usual, prefetching those other attributes is cost-beneficial if some of them are referenced later for at least several other objects in S.

If S is a set of objects and AttrName is a many-to-many relationship, then the references from O to objects in S must be stored in a junction table T_R, as shown in Fig. 6. In this case, attributes of D are not stored in the same table as the relationship. Retrieving them requires an extra join, so, as in Case (i), they should be prefetched only if there's a high probability of subsequent access.

Case (vi) If AttrName is set-valued (i.e., a set of scalars or objects) and the context of O is a set S′, then prefetch the AttrName set for every object in S′. For example, if AttrName is a relationship R to a set of objects, then execute one SQL statement that prefetches ObjectIDs of members of the R set of every object in S′. That is, the SQL statement returns a table of $\langle \mathrm{ObjID}_1, \mathrm{ObjID}_2 \rangle$ pairs, where ObjID_1 denotes an object in S′ and ObjID_2 denotes an object referenced by ObjID_1 via R. This is analogous to Sect. 2.4, where the scalar attributes of all objects in S′ are prefetched. In addition, for each object O_i in S′, for each $O_{i,k}$ in O_i's R set, assign O_i's R set to be the context of $O_{i,k}$. The prefetch is cost-beneficial if those R sets are accessed for at least several other objects in S′.

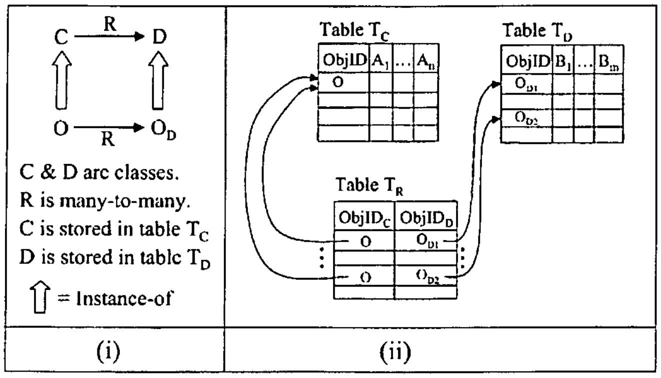

Fig. 6. Prefetching referenced objects in a many-to-many relationship

If R is one-to-many from O, then Fig. 5 can be viewed as an example of this case. R is one-to-many from O's class C to class D, and is represented as a foreign key in D's table T_D. However, instead of retrieving AttrName only for O (as shown in Fig. 5), retrieve AttrName for all objects in O's context. As in earlier cases, the attributes of objects in D can be prefetched too, modulo the additional cost, depending on whether FKey is part of the compound index on T_D.

Similarly, if R is many-to-many, then Fig. 6 can be viewed as an example of this case. Again, instead of retrieving AttrName only for O, retrieve AttrName for all objects in the context of O.

ExecuteQuery(Q) – We apply context-based prefetch to the set that results from the execution of a query, just as we do for a set of stored object references in Cases (iii) and (iv). We could do this by executing the query, saving the resulting ObjectID's in a set, and sending the ObjectID's to the RDBMS on each prefetch. However, this is inefficient for large query results, and problematic since the query that does the prefetch could exceed the maximum size of a SQL statement or stored procedure call. We could re-execute the query on each prefetch, but this too would be inefficient unless the query is cheap. Therefore, we store the context on the server in a temporary table.

So, assume each session has a temporary table TEMP (SetID, ObjectID) in the DBMS. Associate a unique SetID, s, with the query. Map the query into an Insert statement that appends rows $\langle s, o\rangle$ to TEMP for each ObjectID o in the result of the query. Execute the Insert and return the ObjectIDs of the query result into a set, S, also identified by s.

Creating this context is cost-beneficial if attributes of the objects in S are subsequently accessed, making prefetches of those attributes cost-beneficial, and if the query is too expensive to re-execute to perform the prefetch.

TEMP can be created any time after starting the database session, but no later than the first call to ExecuteQuery.

Like any query result, a query context in TEMP can be used across transaction boundaries only if the application is using read-committed, not repeatable-read, isolation.

S.GetNext – Return the designated element of S. Prefetching was accomplished when S was loaded, i.e., in GetObject(ObjectID) or O.GetAttribute(AttrName).

2.6 Managing structure context

Structure context is part of the state of a loaded object. The context's lifetime is governed by that of the object. Thus, when the object is released, the context is deallocated too. This includes context information that is maintained with the object by the OMRDB, typically in main memory. Also, when releasing a set that was the result of an ExecuteQuery, it includes the deletion of rows of the temporary table containing the cached result of the query. Like any deallocation, the latter can be done lazily and asynchronously with respect to other processing.

As for persistent database tables, the performance characteristics of temporary tables must be carefully analyzed when using them to cache query results. For example, depending on how space is managed, it may be important to preallocate temporary table space. It may or may not be valuable to have an index on SetID, depending on how queries are processed, the size of query results, and how many query results are concurrently active.

Suppose an object is loaded multiple times via different navigational paths. For example, an application might load object O via O′.GetAttribute(AttrName), where AttrName is a set S containing O, and later reload O via GetObject(o), where o is O's ObjectID. In some programming interfaces, the later operation does not return the same loaded object, but rather returns another copy of the same persistent object. If so, then one would expect the two objects to share their cached persistent object state, since they refer to the same object. However, since the two objects were loaded via different paths, they should have different contexts. In the example, the first object's context is S, while the second's is null.

2.7 Avoiding repetitious prefetch

Blindly applying the prefetches described in the case analysis of Sect. 2.5 risks prefetching object state that is already in cache. For example, Case (iv) recommends prefetching the attributes of all objects in an object O's context, C, whenever attributes of O are requests. If many of the objects in C already have their attributes in cache, then blindly prefetching these objects' attributes is wasted effort. The wasted effort includes both the retrieval of more data than is needed and the client processing needed to load that data into cache.

The OMRDB can avoid this wasted effort by first identifying which objects in C already have their attributes cached and then only issuing a prefetch for the objects in C whose attributes are not cached. Unfortunately, the benefit of avoiding wasted prefetches entails the following two costs: First, enumerating and checking for cached state of O's context is extra work. One might expect this enumeration is needed anyway, to construct the SQL query that does the prefetch for all objects in the context of O. However, this is usually not the case, because most contexts have a representation of their content that is more compact than a simple enumeration. In particular, if the context of O is a collection that was constructed by dereferencing an attribute A of a parent P of O, then the context can be represented by the pair [P.ObjectID, "A"]. Second, using an explicit enumeration of the context leads to more verbose SQL statements to perform the prefetch. That is, the SQL query needs to include the list of ObjectIDs of the objects that are the subject of the prefetch, rather than using the more compact representation, such as [P.ObjectID, "A"], to perform a simple join, as shown in Sect. 2.5 Case (vi) and the accompanying figures. If the context is large, this potentially can bump into restrictions on maximal size of a SQL query on some systems. In addition, overhead associated with executing a long query string may mitigate some of the benefit of retrieving and processing a smaller amount of data. Still, we expect the benefits usually outweigh these costs, making it worthwhile to identify cached state before prefetching.

Many-to-many relationships are a common case where data to be prefetched may already be in cache. Consider a set of parent objects $P_1, \ldots, P_m$, each of which has a relationship to n children, $C_{i,1}, .C_{i,n}$ (for parent P_i), where each child object may be shared by multiple parents. Suppose all parents and children are loaded into cache, but the attributes of the children are not yet loaded. (Notice that this is an example of the scenario of the previous section, where an object (i.e., child) is loaded multiple times, each loaded instance having a different context.) Using context-based prefetch, an access to an attribute of a child $C_{i,j}$ triggers a prefetch of that attribute for all children in the context of $C_{i,j}$, where that context is its parent's collection of children. If children of many different parents are fetched, then the probability of repeatedly prefetching the same children can be high, potentially defeating the benefit of the prefetch. We will see an example of this in Sect. 5.1.

3 Performance hints

Since the prefetch optimizations of Sect. 2.5 are not always cost-effective, it is important to be able to enable and disable them. Enabling an optimization is essentially a performance hint to the underlying OMRDB. This general approach of application-supplied hints has been adopted by other object-to-relational mapping systems, such as [23], and in other commercial products [8].

The main optimizations that are worth controlling in this way are the following:

1A. Attributes for 1 object – When accessing an object O, prefetch all of O's scalar attributes.

1R. Relationships for 1 object – When accessing an object O, prefetch all of O's references to other objects.

MA. Attributes for Many objects – When accessing a scalar attribute A of an object O, prefetch all of the attributes in A's table for all objects in O's set context.

MR. Relationships for Many objects – When accessing a relationship attribute A of an object O, prefetch the objects related via A for every object in O's set context.

It is often appropriate to enable a prefetch optimization for an application's entire execution, but sometimes it is not. For example, if an application accesses all members of some sets but only one member of others, MA will speed up accesses to the former and slow down accesses to the latter. Thus, it is beneficial to enable and disable the optimizations dynamically during its execution.

Performance hints can be added as tags in an information model to specify the default prefetch behavior of a class. For example, a class C could be tagged to enable MR but disable MA. Such default behavior can be overridden by the application.

Although prefetch is enabled and disabled dynamically, it is still beneficial to maintain context for each loaded object. This makes it possible to perform context-based prefetch on an object that was loaded when prefetching was disabled. Since maintaining context is cheap, there is little benefit to disabling it, with one possible exception: The result of a query is cached in a temporary table for use as the context of each object in that result. Since adding the result to the temporary table has non-negligible cost, it is probably worth disabling in cases where it is known to be ineffective.

Although the manual control that performance hints offer is worthwhile, it would be even better if the system could automatically decide when to enable and disable prefetching. One approach is to run the application on sample data to generate traces, and then analyze the traces to determine whether each type of optimization is cost effective for a given run. Another approach is to statically analyze the application to predict certain access patterns. For example, a common pattern is to call GetAttribute(R), where R is a relationship that returns a set of object references, and then loop through the result, accessing attributes of each object. In this case, the program would be modified to enable MA before entering the loop. Even finer-grained control is possible here, since the exact set of attributes that will be referenced could be made known in the hint, thereby reducing the cost of the prefetch.

A cache control mechanism would also be beneficial, to reduce the amount of prefetching when the OMRDB's cache is stressed. The cache manager could track the amount of free space, for this purpose.

4 Implementation in Microsoft Repository

The prefetch optimizations of Sect. 2 are implemented in Microsoft Repository 2.0, whose storage engine is a persistent object layer implemented on top of Microsoft SQL Server 7.0. The product's object model, table layout, and API do not include all of the alternatives in Sects 2.1–2.3. The differences that are relevant to the prefetch optimizations being considered here are:

- All relationships are currently stored in a single junction table, such as Table J in Fig. 3(i), not as foreign keys in class-oriented tables.

- The object model is COM, where classes implement interfaces, and interfaces have single-valued scalar properties and relationship-valued collections. Each interface's scalar properties are stored in (i.e., are columns of) one table, each of whose rows contain state for an object whose class implements the interface. Each table can store the properties of many interfaces, but a class's interfaces need not all be stored in one table.
- There is an additional method, ObjectInstances, that gets the set of all objects that are instances of either a given class or a class that supports a given interface. Objects that are retrieved by this method have the retrieved set as their context, which is represented by the identity of the class or interface. This is very similar to Case (v) in Sect. 2.5, O.GetAttribute(AttrName), where AttrName designates a set of object references.
- Each time a persistent object P is loaded, a new COM object is created, which shares its cached state with other COM objects that represent P. The context is stored in the COM object, since each load operation for P may be via a different navigational path and therefore have a different context to guide prefetch.

The object model, table layout, and API are described in detail in [1,2].

5 Performance measurements

The prefetch optimizations as implemented in Microsoft Repository have been useful for many customer applications we have tested, so they are frequently enabled – especially MA, which we have found is almost always beneficial. However, it is hard to report on these in a way that bears scientific scrutiny, since each application yields a varying workload that is hard to characterize succinctly. We therefore ran some more controlled experiments to show how the optimizations work in practice.

5.1 Experiment 1 – the OO7 benchmark

A good test to show the benefits of MA is the OO7 benchmark [4,5,6], since it is a highly regular workload that is representative of many persistent object applications (a brief summary is given in the Appendix). We ran the OO7 queries and traversals using Microsoft Repository on the medium-sized OO7 database (225MB). We ran with a cold server cache, to ensure the optimizations are fully penalized for useless prefetches. We did not run OO7 structural modifications, since their behavior is not affected by our optimizations.

Queries Q1–Q3 and Q7 are all of the following form: retrieve a set of objects (in our case using ExecuteQuery) and access the objects' state. Q8 also accesses state, to form a sub-query for each object in an outer query (it retrieves a set of object pairs). So they all benefit from optimization MA, which prefetches the objects' state in one round-trip based on the cached query result. Fig. 7(i) reports the percentage improvement in running time over an execution with no prefetch optimizations enabled (i.e., [(old – new)/old] × 100). The benefit varies based on the size of the result and the algorithm for joining the TEMP table (containing cached query results) with the attribute table at the server. Query Q4 is of the same form as Q1-Q3, but the query plan for the prefetch is sub-optimal, making MA ineffective. None of the queries benefit from 1R or MR, because they don't access any relationships; neither do they benefit from 1A, because only one attribute of each object is accessed. Query Q5 simply counts the number of objects in the result of a query, so there's no benefit to prefetching properties or relationships.

007 Test	Benefit of MA
Q1	87 %
Q2	17 %
Q3	54 %
Q4	-39 %
Q5	0 %
Q7	73 %
Q8	42 %

(i)

007 Test	Benefit of MA
T1	1 %
T2a	-11 %
T2b	34 %
T2c	31 %
T3a	-12 %
T3b	34 %
T3c	39 %
T6	0 %

(ii)

Fig. 7. Benefit of prefetch in OO7

Traversals T1–T6 navigate relationships starting from a root. Each test traverses the three subassemblies of each assembly, down through six levels. For each leaf assembly, called a *base assembly*, it accesses the three composite parts connected to the base assembly. It then accesses one or more of the atomic parts connected to each composite part (the type of atomic part accesses differ for each traversal).

Like the queries, traversals T2b,c and T3b,c access attributes and therefore benefit from MA. T1 and T6 do not access attributes, so MA has no effect. T2a and T3a access only one object in each atomic parts collection, so prefetching attributes for all objects is a cost that has no compensating benefit. Thus, MA decreases their performance.

In principle, traversals T1–T3 should benefit from MR. To see why, consider a parent assembly, P, that has three child subassemblies, C1, C2, and C3. Without MR, the subassembly relationships of C1, C2, and C3 are expanded independently, requiring three round-trips. With MR, when C1's subassembly relationship is referenced, the subassembly relationships of C1, C2, and C3 are fetched in one round-trip, since C2 and C3 are in C1's context (namely, the subassembly collection of P). Thus, three round-trips to the database server are replaced by one. When we ran test T2b, for example, we observed a 60% reduction in the number of round-trips, in line with expectations.

However, there is a mitigating factor involving the base assemblies, which are the leaves of the hierarchy. Each of the 729 base assemblies in the OO7 database has a relationship to three composite parts, and each of the 500 composite parts has a relationship to 200 atomic parts. Since there are 729 × 3 = 2187 relationships from base assemblies to composite parts, and only 500 composite parts, most composite parts are reused in multiple base assemblies. Since we did not eliminate repetitious prefetch (as described in Sect. 2.7), shared components caused MR to repeatedly prefetch the same composite parts many times. The client cost of needlessly reconstructing 2187 – 500 = 1687 atomic parts collections of these already-cached shared composite parts washed out any benefit of the reduced number of SQL queries. Thus, with the current feature set of our implementation, OO7 does not benefit from MR.

5.2 Experiment 2 – XML export

Another application with a highly regular workload is the XML export utility that ships with the product. We wrote a program that uses the utility to export a set of objects reachable from a root, by doing a breadth-first traversal from the given root and writing them out as an XML stream. We exported a UML model consisting of 3100 objects and 3900 relationships. The execution time was 267 ms with no optimizations, 117 ms with 1R enabled, 220 ms with MA, and 78 ms with both 1R and MA. That is, the execution was 56%, 18%, and 71% faster with optimizations 1R, MA, and 1R+MA, respectively.

1R helps because in UML each object has many relationship types, all of which are accessed, either to export them or determine that they are empty. MA helps because if a relationship is non-empty, then attributes of all related objects are exported. These benefits are nearly independent, in that the benefit of 1R+MA (71%) is nearly the sum of the benefits of 1R and MA (56% + 18%=74%).

5.3 Experiment 3 – Measuring MR

To measure the benefits of MR in a controlled setting, we created an object hierarchy stored as relationships in a junction table like Table J of Fig. 3(i), with a clustered index on its key (Source, RelType, Target). The hierarchy has a top-level fan-out of 100, and a second-level fan-out of 20. We traversed the hierarchy by running SQL. The baseline ran a SQL query to get the 100 children of the root followed by a query for each child to get its 20 children. To model MR, we replaced the 100 queries for grandchildren by one query to get all grandchildren. The latter retrieves the same amount of data as the former, but replaces the fixed overhead of 100 queries by that of just one query. The resulting execution time was 71% faster with MR enabled than disabled.

MR is sensitive to the size of the context to which it is applied, since that affects the number of queries it saves. For example, by running the previous experiment on a hierar- chy with a top-level fan-out of 20 and a second-level fan-out of 100, the benefit of MR is only 33%, less than half as much as the previous case. The same amount of data is retrieved as before, but only 20 queries for grandchildren are replaced by one query. Similarly, in XML, MR actually reduces performance, probably because the contexts are small and the prefetch requires an extra expensive join.

6 Extensions

6.1 Asynchronous prefetch

Some contexts are large, causing MA or MR to prefetch a large amount of data. Performing this prefetch synchronously delays the application's data access that triggered the prefetch. This delay is especially worrisome when communication between the OMRDB and database server is slow. However, even with high bandwidth communication, the delay incurred by a large prefetch can be significant.

To some extent, an application can reduce this delay manually by reordering its logic so that the OMRDB gets data before the application actually needs it. However, this is a major programming burden and cannot always be accomplished.

It would be better to have the system reduce the delay by doing the minimal work necessary to return from each data access call and prefetch any additional data asynchronously with respect to the call. This not only helps reduce application delay, but is also useful when the cache is nearing overflow. The ability to delay part of the prefetch allows the OMRDB to postpone loading the data until enough cache space is freed to accommodate the prefetched data.

Asynchronous prefetch can be done in the OMRDB without help from the underlying DBMS by partitioning the query that performs the prefetch into two queries: one that retrieves the data required by the application's data access plus, possibly, a subset of the data to be prefetched, and a second remainder query that identifies the rest of the data to be prefetched. Unfortunately, this isn't always doable. For example, in MA, to prefetch attribute values for a large unordered set, we would need a column value that partitions the set into non-overlapping subsets, something that we cannot count on in general.

We can do better by running the prefetch as a single query and leveraging asynchrony offered by the underlying SQL DBMS. A SQL query returns its result as a set of rows. Using low-level interfaces offered by the data access protocol (e.g., via ODBC), the OMRDB can start populating its cache as soon as the first packet arrives from the DB server. When the OMRDB has cached all of the data requested by the application, it returns to the application, but it continues processing packets asynchronously until all the prefetched data has arrived. If the application later asks for an item that is still being prefetched, it is blocked.

Ordinarily, as long as a query is active on a database session (connecting the OMRDB to the SQL DBMS), the OMRDB cannot issue additional queries. Thus, if the application makes a data access call that requires an SQL call to the DBMS, it may have to wait while the OMRDB finishes prefetching the results that were triggered by a previous call.

To circumvent this problem, many DBMS products, such as DB2, Oracle, and Microsoft SQL Server, support multiple active statements on a database session. This allows a second application request to be executed while a previous request is still active. To meet our needs, this feature must allow the result of a query to be buffered on the DB server until it is explicitly requested by the OMRDB client. We call this feature *server cursors* (which is the name used for it in Microsoft SQL Server). Using server cursors, the OMRDB can buffer the results of several queries on the server. Therefore, it can execute a query that performs a prefetch (saving its result in a server cursor), load part of the prefetched data into cache, and then turn its attention to servicing other application requests or prefetches before returning to finish processing the original prefetch.

If server cursors are used and the application abandons the object that was the subject of the prefetch, the OMRDB may never have to finish loading the prefetched data. For example, suppose we are applying MA to an object O in context C, where C is a collection. The prefetch retrieves attributes of all objects in C. If the application releases its handle on C before the prefetch is complete, then the prefetch can stop and the cursor is dropped.

If server cursors are not available, an alternative is to use two sessions for each transaction: one session to run synchronous requests and a second session for asynchronous prefetches. However, like multiple statements per session, this feature is not universally supported. Some DBMSs do not allow two sessions from one client to be running the same transaction. The main reason for this restriction is that it would require adding server support for concurrent nested transactions, i.e., the server must be able to independently back out update statements on each session.

Streaming the result of a query is useful for ordinary fetch, not just prefetch, such as when getting the objects in a large collection. However, in some cases, it is unavoidable to fetch the entire result before returning to the caller. For example, Microsoft Repository stores sequenced collections as rows linked by back pointers [1]. If an application requests an item of the collection other than the first (the first element can be distinguished by its NULL back pointer), then all of the collection elements must be loaded into cache for the engine to determine the requested item.

Finally, statistics are needed to determine how much to prefetch. The goal is to fetch only a small amount to avoid delaying the application much by the prefetch, but to fetch enough to keep the application busy while the second, asynchronous prefetch is running.

Some contexts are too large to prefetch the attributes or relationships for every object in the context. For this case, one could specify a threshold for context size, above which the MA or MR prefetch is disabled.

6.2 Prefetching across paths

We can extend context-based prefetch to apply to paths of relationships. For example, consider a persistent object base that contains a database schema definition object, which contains table definition objects, each of which contains column definition objects, each of which is related to a data type definition object. When accessing a column of the first table definition, MR will prefetch column definitions of all table definitions. We showed in Sect. 5.3 that this can yield significant improvements. It may also be beneficial to prefetch the scalar attributes of those column definitions [i.e., Sect. 2.5, Case (v)] and their relationships to data type definitions (adding another hop to the path). To estimate the benefit, recall in Sect. 2.4 that getting 100 rows from two identical tables with 100-byte rows in 2 round-trips is 19% slower than getting the join in one round-trip. One could extend the hints of Sect. 3 to specify when to prefetch across such a path.

An issue with this approach is the representation of the inherently hierarchical result of the prefetch, when a relational query is used. As observed in [19], in a parent-child relationship, when retrieving all the children of a parent, the parent information is repeated for every child due to the normalization inherent in relational queries. Possible solutions are to return the result in a nested table, a tree structure, or an XML stream (that represents the tree structure), all of which require some extension to the underlying RDBMS.

6.3 Lazy loading of sets

Suppose O.GetAttribute(AttrName) returns a set, S, of objects [i.e., Case (v) and Case (vi) in Sect. 2.5]. One could execute GetAttribute by storing only its definition [i.e., "O.GetAttribute(AttrName)"] and not its instances in main memory. If S is used only to insert new objects, then the existing state of S never needs to be loaded, a significant optimization. If existing objects in S are retrieved, then its instances can be loaded on the first invocation of S.GetNext.

Suppose it is known that all objects in S are instances of the same concrete class C (rather than different specializations of C). Now even GetNext can avoid loading an instance of S, by creating a hollow instance O′ of C. This can be done using only cached type information, without accessing database instances. The state of O′ is populated only when one of its attributes is accessed. At this point, MA kicks in; it retrieves that attribute (and possibly others in the same table) for all objects in S. This prefetch gets the object IDs of all objects in S as a side effect, which allows S finally to be populated with instances. Thus, the initial round-trip to the RDBMS to get the object IDs of objects in S is entirely avoided, leaving only one round-trip to get the attributes of all objects in S; the number of round-trips in this scenario is halved.

While this benefit is appealing, unfortunately the line of logic to attain it is not quite sound: If S is empty, then S.GetNext should return null. If this is known to be impossible (e.g., because the set has an enforced integrity constraint saying it has non-zero cardinality), then the technique works fine. Otherwise, if it can return null, then it is not valid for it to return a hollow instance of C. Therefore, to benefit from this optimization, a change is needed in the programming interface. There are several options: a hint could be issued before the first call to GetNext, to tell the system what attribute(s) to prefetch; the GetNext call itself could optionally include a list of attributes of interest; or the semantics of GetNext could be modified so that the first GetNext on an empty set returns an object and an exception is raised only when attempting to access one of that object's attributes. The use of a hint strikes us as the best of the alternatives.

7 Related work

Although much has been published on the implementation of persistent objects, very little of it uses an RDBMS as the underlying store. Keller et al. provide a good overview of the issues [16]. Most papers, such as [18,19,21,27], assume that the set of objects to be retrieved is defined by a query, such as [18, 19, 21, 27], where the issues are running the query efficiently and assembling the objects in the OMRDB, and caching the query result for reuse with later queries [15]. Navigational access is applied to the result of the query, so there's nothing to prefetch. Descriptions of some commercial products that map objects to relations can be found in [22,26,28,31].

Proponents of OODBs have published many white papers to show that their products outperform a similar implementation on RDBMSs, but little of this has made it into the scientific literature. A useful bibliography is [9].

Prefetching architectures based on recent reference behavior are described in [12,24]. Palmer and Zdonik use pre-analyzed reference traces to guide prefetch [24]. During a training phase, they look for recurring access patterns. Popular patterns are stored in an associative memory and used to drive prefetch during normal operation. Curewitz et al. propose using compression algorithms to find recurring access patterns [12]. Compression algorithms work by determining the probability that certain sequences will occur and encode those sequences in a small number of bits. Curewitz et al. treat object references as an alphabet over which reference traces (i.e., sequences) can occur. Given a reference trace, they use the compression algorithm to predict which references are most likely to occur next and use those predictions as candidates for prefetch.

Both [12] and [24] look for recurring access patterns. By contrast, our approach assumes that applications conform to a certain generic navigational pattern, so programs benefit from prefetch even the first (and possibly only) time the objects are accessed. Since the generic access pattern is not always relevant, we allow application programmers to influence prefetch decisions. To avoid this burden, one possible application of [12] and [24] in our setting is to use reference traces to predict when context-based prefetching should be enabled. They could also be used to determine whether data in multiple tables is frequently accessed together and should therefore be prefetched in one round-trip. However, it is possible that simpler analyses of reference traces than these rather sophisticated techniques would yield similar benefits.

Haas et al. investigate ways of prefetching object state along with object IDs, the same problem as this paper [14]. However, their approach is based on two assumptions that do not hold in our environment: first, that query processing can be distributed between the database server and the client that submits the query (in their case, a middle tier server), and second, that the query optimizer can be extended to be knowledgeable about the caching activity. They include the caching activity as a possible step of a query plan. Each caching step has a cost, which is the actual cost to cache the data minus the benefit of the cached result to reduce the cost of later object accesses. They use this cost during conventional cost-based query optimization to decide whether to do caching and, if so, how much of the query evaluation to do on the server.

In our case, we face a very similar problem when prefetching the result of ExecuteQuery. We assumed we could not modify the query optimizer, so we always retrieve the object IDs first and retrieve the object state in a separate step. To avoid recalculating the query, we cache the object IDs on the server. If the tables containing these objects' state is still in server cache when we later access them, then the cost of reassembling the object state should not be much different than that incurred by Haas et al.'s approach. If not, then Haas et al.'s approach could beat our ExecuteQuery performance, since it gathers up the state while the required tables are in cache. However, this benefit comes at a substantial engineering cost of implementing a query processor for the client and of integrating the caching mechanism tightly with the query processor, which reduces the portability of the OMRDB. Moreover, their approach is probably not helpful in the case of prefetching simple objects and relationships. Such prefetches are performed by simple queries, which can be statically analyzed during OMRDB implementation for the best distribution of function between client and server.

Prefetching also arises in database-driven Web sites, where the contents of a page is the formatted output of the execution of a query. A general discussion of Web-oriented prefetching techniques is in [25]. A particular technique that is similar to ours appears in Florescu et al. [13]. They consider a variety of optimizations for computing queries that generate Web pages, some of which involve prefetch. In their model, a Web site's content consists of a set of pages (i.e., query results), which are represented by nodes of a graph. Navigating from one page to another involves executing a query, which is the label of the edge connecting the two nodes. Frequently, the query q that navigates from one node to another is related to queries executed for earlier pages that the user requested. For example, the result of a previous query q' may either be a sub-query of q or have significant overlap with q. In this case, when evaluating q', it can be beneficial to retrieve and cache extra data that can speed up the later evaluation of q. This retrieval of extra data amounts to a prefetch. Florescu et al. show how to derive a query q'' from q and q' and how to use the cached result of q'' to evaluate q and q', which they call *lookahead computation*. One variation, called *optimistic lookahead*, is strikingly similar to MA and MR; when executing q', they also prefetch the data needed for the next query q for all possible bindings of q''s parameter. An example of this in our scenario is prefetching all relationship collections reachable from the objects in a given collection context. Another example is caching the result of ExecuteQuery(q') to reuse it when issuing a query q for the state of objects returned by q'.

Comparing the scenario of [13] to the problem of this paper, it is as if the application that uses our OMRDB were represented as a data-flow graph, where the edges are queries issued by the OMRDB in response to application requests, such as ExecuteQuery, GetObject, and GetNext. Then, one could use the lookahead techniques of [13] to reuse the (partial) results of earlier queries when evaluating later ones. The question is: Will successive queries issued by tools against an OMRDB generate overlapping queries that could benefit from lookahead computations? In the Web scenario for which these lookahead computations were developed, such overlap is quite common, since users progressively narrow their search as they navigate through a sequence of pages. Whether such overlaps occur frequently for arbitrary applications that use an OMRDB is more problematic and could only be determined through analysis of typical workloads. Such overlaps would need to be very common to justify the substantial engineering cost of applying the lookahead technique to OMRDB applications, which would require analysis of the application to extract the data flow graph, in addition to the optimization work described in [13].

As mentioned in the Sect. 1, clustering objects on pages in a POODB amounts to a static prefetching algorithm. A more dynamic version of this approach is the hybrid adaptive caching algorithm (HAC) used in the THOR system [20]. In HAC, each fetch is for an entire page. However, when a page is selected for replacement, popular objects in that page are copied to an object cache before the page is evicted from the page cache. Thus, each page-oriented fetch can be viewed as a prefetch of a collection of objects, where prefetching errors are corrected when the page is evicted.

8 Conclusion

We described a technique for predicting useful prefetches when a navigational object-oriented interface is implemented on a relational DBMS. We presented a design for the technique and measured its performance in a commercial product, Microsoft Repository 2.0. We proposed a number of extensions, some of which would benefit from further work, such as automatically issuing hints to enable the prefetch optimization and prefetching across paths of relationships.

Overall, there has been much published about efficient implementations of persistent objects. Having worked on an implementation of persistent objects on a relational database for the past several years, we feel that the problem of optimizing the performance of such a system is only partially understood and would benefit from much more research. Given the advent of object-relational DBMSs, and the need to offer persistent object interfaces on top, the importance of this problem is growing.

Appendix – Summary of OO7

The OO7 benchmark is based on a bill-of-materials database. In the medium-sized database, there are 500 composite parts, each of which has an associated document and 200 interconnected atomic parts (i.e., 100 000 atomic parts in all). There is also a seven-level assembly hierarchy, starting from a single root. Each non-leaf assembly has three subassemblies, so there are 729 (= 3^6) leaves, which are called base assemblies. Each *base* assembly has 3 randomly selected composite parts. The following queries and retrievals are taken from [6], paraphrased to save space:

Q1 – Given 10 random atomic part IDs, get the atomic parts (that exist) and the number retrieved.

Q2 – Given a range of dates containing the last 1% of dates in atomic parts, retrieve the atomic parts.

Q3 – Given a range of dates containing the last 10% of dates in atomic parts, retrieve the atomic parts.

Q4 – Given 100 random document titles, for each document, find all base assemblies that use the composite part corresponding to the document. Also return the number of such base assemblies.

Q5 – Find all base assemblies that use a composite part whose build date is later than that of the base assembly. Also report the number of base assemblies found.

Q7 – Scan all atomic parts.

Q8 – Find all pairs of documents and atomic parts where the atomic part's document ID equals the document's ID. Return the number of pairs found.

T1 – Traverse the assembly hierarchy. For each base assembly, visit its composite parts. For each composite part, do a depth first search on its atomic parts. When done, return the number of atomic parts visited.

T2 – Same as T1, but swap attributes x and y for some of the objects:
a. Update one atomic part per composite part.
b. Update every atomic part encountered.
c. Update each atomic part in a composite part four times.

T3 – Same as T2, but update the (indexed) date field.

T6 – Same at T1, but for each composite part, visit only its root atomic part.

Note that there is no Q6, T4, or T5 in OO7. Traversals T7 and T8 are omitted because they run very fast and therefore lead to inaccurate (high variance) measurements.

Acknowledgements. We thank Thomas Bergstraesser and Jason Carlson for many stimulating discussions about the performance of Microsoft Repository. We are also very grateful to Jayaram Mulupuru and Daniel Reib for implementing the approaches described in this paper and to Jim Hance and Gary Miao for measuring them, all of which helped uncover many of the detailed problems we addressed.

References

1. Bernstein P.A., Harry B., Sanders P.J., Shutt D., Zander J. The Microsoft Repository, Proc 23rd VLDB Conf, 1997, pp. 3–12
2. Bernstein P.A., Bergstraesser T., Carlson J., Pal S., Sanders P.J., Shutt D. Microsoft Repository Version 2 and the Open Information Model, Information Syst 24(2): 71–98, 1999
3. Booch G., Rumbaugh J., Jacobson I. The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, 1998
4. Carey M.J., DeWitt D.J., Naughton J.F. The OO7 Benchmark. Proc 1993 ACM SIGMOD Conf, pp. 12–21
5. Carey M.J., DeWitt D.J., Kant C., Naughton J.F. A Status Report on the OO7 Benchmark. Proc OOPSLA 1994, pp. 414–426
6. Carey M.J., DeWitt D.J., Naughton J.F. The OO7 Benchmark, technical report. Univ. of Wisconsin, January 1994, ftp.cs.wisc.edu
7. Cattell R.G.G., Barry D., Bartels D., Berler M., Eastman J., Gamerman S., Jordan D., Springer A., Strickland H., Wade D. The Object Database Standard: ODMG 2.0. Morgan Kaufmann Publishers, 1997
8. Chang E.E., Katz R.H. Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS. Proc 1989 ACM SIGMOD Conf, pp. 348–357
9. Chaudhri A.B., ODBMS Resources. http://www.soi.city.ac.uk/ akmal/html.dir/resources.html
10. Cluet S. Designing OQL: Allowing Objects to be Queried. Information Syst 23(5): 279–306, 1998
11. Copeland G., Maier D. Making SmallTalk a Database System. Proc 1984 ACM SIGMOD Conf, pp. 316–325
12. Curewitz K.M., Krishnan P., Vitter J.S. Practical Prefetching via Data Compression. Proc 1993 ACM SIGMOD Conf, pp. 257–266
13. Florescu D., Levy A., Suciu D., Yagoub K. Optimization of Run-time Management of Data Intensive Web Sites. Proc 25th VLDB Conf, 1999, pp. 627–638
14. Haas L.M., Kossmann D., Ursu I. Loading a Cache with Query Results. Proc 25th VLDB Conf, 1999, pp. 351–362
15. Keller A., Basu J. A Predicate-based Caching Scheme for Client-Server Database Architectures. VLDB Journal 5(1): 35–47, 1996
16. Keller A., Jensen R., Agrawal S. Persistence Software: Bridging Object-Oriented Programming and Relational Database. Proc 1993 ACM SIGMOD Conf, pp. 523–528
17. Lamb, C., Landis G., Orenstein J.A., Weinreb D. The Object-Store System. CACM 34(10): 50–63, 1991

18. Lee B.S., Wiederhold G. Outer Joins and Filters for Instantiating Objects from Relational Databases Through Views. IEEE Trans Knowl Data Eng 6(1): 108–119, 1994
19. Lee B.S., Wiederhold G. Efficiently Instantiating View-Objects From Remote Relational Databases. VLDB Journal 3(3): 289–323, 1994
20. Liskov B., Castro M., Shrira L., Adya A. Providing Persistent Objects in Distributed Systems,. Proc ECOOP 99, pp. 230–257
21. Mitschang B., Pirahesh H., Pistor P., Lindsay B.G., Sdkamp N. SQL/XNF – Processing Composite Objects as Abstractions over Relational Data. Proc 1993 Int Conf Data Eng, pp. 272–282
22. Ontos, http://www.ontos.com
23. Orenstein J.A., Kamber D.N. Accessing a Relational Database through an Object-Oriented Database Interface. Proc 21st VLDB Conf, 1995, pp. 702–705
24. Palmer M., Zdonik S. Fido: A Cache that Learns to Fetch. Proc 17th VLDB Conf, 1991, pp. 255–264
25. Palpanas T. Web Prefetching Using Partial Match Prediction, Technical Report CSRG-376. Department of Computer Science, University of Toronto, 1998, http://www.cs.toronto.edu/csri.
26. Persistence Software, http://www.persistence.com
27. Pirahesh H., Mitschang B., Sdkamp N., Lindsay B.G. Composite-object views in relational DBMS: an implementation perspective. Information Syst 19(1): 69–88, 1994
28. POET Software, POET SQL Object Factory, http://poet.com/factory.htm
29. Rational Software Corp., Unified Modeling Language Resource Center. http://www.rational.com/uml
30. Rogerson D. Inside COM. Microsoft Press, 1997
31. RogueWave Software, DBTools.h++, http://www.roguewave.com/products/dbtools/